




Stacks and Queues

WEDNESDAY, APRIL 9

Today we will discuss use of **Stack** and **Queue**. These containers store data in an ordered format and are used to solve many problems.

-  Readings: [Text](#) 5.1 - 5.3, Class documentation for [Stack](#) and [Queue](#)
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

Lecture Video

Click to sign in and play video

Contents

1. Announcements
2. Attachments
3. Preliminaries: Client-Side Approach to ADTs
4. Passing Containers by Reference
5. Stack Overview
6. Stack Examples in Code
7. Stack Applications
8. Vectors as Stacks (and a Note About the Power of Abstraction!)
9. An Important Interlude: `break` Statements
10. Queue Overview
11. Queue Examples in Code
12. Mod Operator (%)
13. Queue Applications
14. Range-Based Loops and Output Streams for Vectors, Grids, Stacks, and Queues
15. **Supplementary** Stack Application: Postfix Notation

16. What's next?

17. Exam Prep

Announcements

- Assignment 1 is due Friday at 11:59 PM. Please be sure to head to LaIR early for help with these assignments (and to ensure that you don't get stuck in a really long queue). We appreciate your patience in LaIR. Please keep in mind we're doing our best to help a large number of students.
- Before section each week, please be sure to see Jonathan's weekly Ed announcement for an indication of what topics/lectures you should be caught up on before going into section.
- Solutions to each week's section problems will be posted to the course website on Friday afternoons. After section each week, be sure to work through any problems you found tricky or that your SL don't have time to cover.
- Your next batch of quizzes will be due next Wednesday at 1 PM.
- Want some help getting oriented to Assignment 2 when it releases this Friday? We have YEAH (**Y**our **E**arly **A**ssignment **H**elp) Hours this Friday, 3-4 PM, in Bldg 160, Room B40. See Bala's [Ed announcement](#) for more information!

Attachments

For ease of access, here are the stack and queue vizualization programs I used in lecture today:

- [StackViz.zip](#)
- [QueueViz.zip](#)

Preliminaries: Client-Side Approach to ADTs

I mentioned again at the top of class today that we're currently taking a **client-side** approach to the ADTs we're exploring. That means we're exploring how to use them, but not their nitty-gritty, behind-the-scenes implementation details. However, we will discuss those implementation details later this quarter, after we've built up some additional requisite knowledge for doing so.

In the meantime, exploring ADTs from the client side empowers us to solve all kinds of interesting problems.

Passing Containers by Reference

(*Key take-away!*) I mentioned this at the end of class today, but I'm putting it at the top of your notes so it doesn't get buried: We often pass containers to functions by reference rather than by value, even if we don't want a function to be able to modify the original container. That's because containers can store huge amounts of data, and passing by value creates an entirely new copy of those containers, which takes time (to copy the elements over) as well as extra space in memory. In contrast, establishing a *reference* to a container is a super fast operation that only has a tiny bit of memory overhead.

For example, the syntax for passing a vector of integers by reference is as follows:

```
void printVector(Vector<int>& v)
{
    ...
}
```

Stack Overview

Attachment: [StackViz.zip](#)

The first ADT we saw today was the stack. A stack is a **LIFO** (last-in, first-out) data structure. It generally supports a limited set of operations:

Member Function	Description
-----------------	-------------

Member Function	Description
clear()	removes all elements from the stack
equals(stack)	returns <code>true</code> if the two stacks contain the same elements in the same order, <code>false</code> otherwise
isEmpty()	returns <code>true</code> if the stack contains no elements, <code>false</code> otherwise
peek()	returns the element on the top of the stack without removing it
pop()	removes the element from the top of the stack and returns it
push(value)	pushes <code>value</code> onto the stack
size()	returns the number of values in the stack
...	For an exhaustive list, see: Stanford Stack class

Notably, a stack does *not* support a search operation or an operation to find and remove an element with a specific value.

Before delving into actual code, we explored the behaviors of stacks through the `StackViz` program attached above.

Stack Examples in Code

To create a stack using the Stanford C++ Libraries, we must include the following:

```
#include "stack.h"
```

The syntax for creating a stack is as follows:

```
Stack< DATA_TYPE > VARIABLE_NAME ;
```

(*Important note!*) We must capitalize the 'S' in "Stack" when declaring one in code. As with the other ADTs we have seen this quarter, C++ has its own built-in version of a stack that uses a lowercase 's'.

Here's the first example we saw of an actual stack in class today:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    Stack<int> s;

    s.push(10);
    s.push(20);
    s.push(15);
    s.push(12);

    while (!s.isEmpty())
    {
        cout << s.pop() << endl;
    }

    return 0;
}
```

output:

```
12
15
20
10
```

We also saw the following, which uses a stack to print the characters of a string in reverse order:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    string str = "muffins";
    Stack<char> myStack;

    for (char ch : str)
    {
        myStack.push(ch);
    }

    while (!myStack.isEmpty())
    {
        cout << myStack.pop();
    }
    cout << endl;

    return 0;
}
```

output:

```
sniffum
```

We also saw that we can print a stack using the << operator. This is super helpful for debugging, and we saw that C++'s built-in stack does not support that behavior:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    Stack<int> s;

    s.push(10);
    s.push(20);
    s.push(15);
    s.push(12);

    cout << "Stack contents: " << s << endl;

    return 0;
}
```

output:

```
Stack contents: {10, 20, 15, 12}
```

(*Not mentioned in class.*) We can also use the == and != operators to check whether two stacks are the same (i.e., they contain the same elements in the same order). For example:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    Stack<int> s1;
    Stack<int> s2;
    Stack<int> s3;

    s1.push(12);
    s1.push(10);

    s2.push(12);
    s2.push(10);

    s3.push(12);
    s3.push(10);
    s3.push(10);

    if (s1 == s2)
    {
        cout << "s1 == s2" << endl;
    }
    else
    {
        cout << "s1 != s2" << endl;
    }

    if (s1 == s3)
    {
        cout << "s1 == s3" << endl;
    }
    else
    {
        cout << "s1 != s3" << endl;
    }

    return 0;
}
```

output:

```
s1 == s2
s1 != s3
```

Stack Applications

Stacks come up all over the place in computer science. Here are a few of the applications we discussed today:

- Stacks are useful for reversing things! See the example above in which we print the characters of a string in reverse order.
- We talked about the **call stack**, which your programs use to keep track of which functions have been called in which order and where we need to return to when we hit the end of some function's execution.
- When you press the "back" button in a web browser, it's using a stack behind the scenes to take you back through your browser history for that particular tab!
- Toward the end of today's notes, there is a supplementary section that talks about how stacks can be used to process arithmetic postfix expressions.

Vectors as Stacks (and a Note About the Power of Abstraction!)

We saw that we could actually implement a stack using a vector. The following two programs effectively have the same functionality:

Stack approach:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    Stack<int> s;

    s.push(10);
    s.push(20);
    s.push(15);

    cout << s.pop() << endl;
    cout << "Stack contents: " << s << endl;

    return 0;
}
```

output:

```
15
Stack contents: {10, 20}
```

Vector approach:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
    Vector<int> v;
    v.add(10);
    v.add(20);
    v.add(15);

    cout << v.remove(v.size() - 1) << endl;
    cout << "Vector contents: " << v << endl;

    return 0;
}
```

output:

```
15
Vector contents: {10, 20}
```

Notice, however, that the stack version is not only easier to read, but also less error prone. There is less code to parse (we can infer its semantics with a quick glance at the operations being performed), and we don't have to stop to consider whether we're pushing and popping at the correct index.

With the vector approach, we might have to stop to consider the relationship between `v.add()` and `v.remove(v.size() - 1)` to ensure we understand what the code is doing. Someone coding this up with vectors could also easily remove elements from the wrong end of the vector or have an off-by-one error that causes us to go out of bounds. Those problems simply don't exist with the stack approach, where the details of the `s.push()` and `s.pop()` operations are abstracted away from us and we can assume they work as intended.

An Important Interlude: `break` Statements

We also talked briefly about `break` statements today.

(*Key take-away!*) A `break` statement causes us to leave whatever loop it resides in. A common idiom is to use a `break` statement to put an end to an otherwise infinite loop (such as a `while (true)` loop), although these can be used to terminate what would otherwise still be a non-infinite loop, as well. When we execute a `break` statement, our program jumps to the line immediately following whatever loop it resides within.

The syntax is simply `break;` on a line by itself. `break` statements are often guarded by conditional statements that cause us to break only when we encounter a specific situation. For example:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
    Stack<char> s1;
    Stack<char> s2;

    string str = "stressed";

    for (char ch : str)
    {
        s1.push(ch);
    }

    // true is always true, so we will loop infinitely unless we
    // trigger the break statement below.
    while (true)
    {
        s2.push(s1.pop());

        if (s2.peek() == 'r')
        {
            // A break statement causes us to break out of whatever
            // loop it resides within. When this is executed, we jump
            // to the line immediately after the loop and continue
            // execution from there.
            break;
        }
    }

    // This is the line we jump to when we execute the break statement.

    cout << s1 << endl;
    cout << s2 << endl;

    return 0;
}
```

output:

```
{ 's', 't' }
{ 'd', 'e', 's', 's', 'e', 'r' }
```

Queue Overview

Attachment: [QueueViz.zip](#)

We then discussed the queue ADT. A queue is a **FIFO** (first-in, first-out) data structure. Like the stack, it generally supports a limited set of operations:

Member Function	Description
clear()	removes all elements from the queue
dequeue()	removes and returns the frontmost element in the queue
enqueue(value)	adds <code>value</code> to the back of the queue
equals()	returns <code>true</code> if the two queues contain the same elements in the same order, <code>false</code> otherwise
isEmpty()	returns <code>true</code> if the queue contains no elements, <code>false</code> otherwise
peek()	returns the frontmost element in the queue without removing it
size()	returns the number of values in the queue
...	For an exhaustive list, see: Stanford Queue class

Before delving into actual code, we explored the behaviors of queues through the `QueueViz` program attached above.

Queue Examples in Code

To create a queue using the Stanford C++ Libraries, we must include the following:

```
#include "queue.h"
```

The syntax for creating a queue is as follows:

```
Queue< DATA_TYPE > VARIABLE_NAME ;
```

(*Important note!*) We must capitalize the 'Q' in "Queue" when declaring one in code. This distinguishes it from C++'s built-in queue, which has a lowercase 'q'. (You're seeing the trend here, yes?)

Here are the first examples we saw of actual queue code in class today:

 **Warning! This doesn't work as intended!**

```
#include <iostream>
#include "console.h"
#include "queue.h"
using namespace std;

int main()
{
    Queue<int> q;

    for (int i = 1; i <= 6; i++)
    {
        q.enqueue(i);
    }

    // This does not actually remove and print all elements from the queue!
    for (int i = 0; i < q.size(); i++)
    {
        cout << q.dequeue() << endl;
    }

    return 0;
}
```

output:

```
1
2
3
```

The following is a more conventional way to loop through a queue while emptying it out. Recall that a coding **idiom** is a good, solid, widely-used approach to accomplishing a particular task.

Idiom #1

```
#include <iostream>
#include "console.h"
#include "queue.h"
using namespace std;

int main()
{
    Queue<int> q;

    for (int i = 1; i <= 6; i++)
    {
        q.enqueue(i);
    }

    // Much better. :)
    while (!q.isEmpty())
    {
        cout << q.dequeue() << endl;
    }

    return 0;
}
```

output:


```
1
2
3
4
5
6
```

The following idiom can be used to loop through a queue while emptying it out (as we did above), but it can also be used to process all the original elements of a queue while adding others to the back:

Idiom #2

```
#include <iostream>
#include "console.h"
#include "queue.h"
using namespace std;

int main()
{
    Queue<int> q;

    for (int i = 1; i <= 6; i++)
    {
        q.enqueue(i);
    }

    // Keep track of original queue size. This loop will only do 6 iterations now,
    // even though we're adding some elements back into the queue as we go.
    int ogSize = q.size();
    for (int i = 0; i < ogSize; i++)
    {
        int val = q.dequeue();
        cout << val << endl;

        // Even values get added back to the queue.
        if (val % 2 == 0)
        {
            q.enqueue(val);
        }
    }

    cout << "Final queue contents: " << q << endl;

    return 0;
}
```

output:

```
1
2
3
4
5
6
Final queue contents: {2, 4, 6}
```

Mod Operator (%)

We briefly saw the mod operator (%) in the above example. Recall that the mod operator returns the remainder after division. So, $5 \% 2 = 1$, since $5 / 2 = 2 \text{ R } 1$. Similarly, $17 \% 3 = 2$ since $17 / 3 = 5 \text{ R } 2$.

Queue Applications

Queues come up all over the place in real-world applications. A few examples from today:

- Purchase queues for online ticket sales, gaming consoles, graphics cards -- you name it!
- Login queues for online games.
- Printer queues for processing multiple print jobs.
- The LaIR queue for processing help requests in CS106A/B!

We will see additional applications of queues later this quarter. There's at least one very common application of queues that we'll see when we get to our section on graph theory; they can be used to implement an algorithm called breadth-first search. A related algorithm, called depth-first search, can actually be implemented with a stack.

Range-Based Loops and Output Streams for Vectors, Grids, Stacks, and Queues

I mentioned today that we can use range-based loops for vectors and grids. For example, if we have a vector `v` and a grid `g`, both of which hold characters:

```
for (char ch : v)
{
    ...
}
```

```
for (char ch : g)
{
    ...
}
```

That style of loop is *not* available to us with the `Stack` and `Queue` classes we discussed today, which generally only give us access to their `top` and `front` elements, respectively.

What we *can* do, however, is print the contents of a stack or queue (or a vector or grid, for that matter) by shipping them directly to `cout`. For example, if we have a stack `s` and a queue `q`:

```
cout << s << endl;
cout << q << endl;
```

The ability to dump the contents of a stack or queue to the screen is super handy for debugging programs that use them! This is one of the things that distinguishes the Stanford `Stack` and `Queue` from the standard C++ implementations (called `stack` and `queue` -- note the lowercase 's' and 'q'), whose contents cannot be dumped to the screen in this fashion. With the standard C++ versions, the two `cout` lines above would give compile-time errors. The Stanford implementations of these ADTs do all kinds of things like this to make your life easier this quarter. 😊

Supplementary Stack Application: Postfix Notation

Here's an additional stack application for your consideration:


When we (as humans) process an arithmetic expression such as "3 + 5 * 2 - 12 / (2 * 3)," we have to respect order of operations, which means that we bounce around to different parts of the expression and potentially make multiple passes through the whole expression as we work to resolve its value.

The arithmetic expression above is presented in what we call "infix notation." The arithmetic operators (+, -, *, /) come *in between* their operands (the numbers upon which they're operating). An alternative form of that expression using "postfix notation" (also sometimes called [Reverse Polish Notation](#)), is as follows:

- Original expression (using **infix** notation): 3 + 5 * 2 - 12 / (2 * 3)
- Equivalent expression (using **postfix** notation): 3 5 2 * + 12 2 3 * / -

With postfix expressions, the given operator comes *after* the operands to which it applies. We can actually process a postfix expression quite easily from left to right, token-by-token, using a stack -- without having to make multiple passes through the expression! The ease of left-to-right processing makes postfix a great representational choice for expressions that are being parsed by a program (like a calculator application).

The stack-based algorithm for processing them is as follows:

 **Warning!** The following algorithm assumes a valid, well-formed postfix expression as its input. There are some gotchas to consider if the postfix expression is not well-formed. (See Exercise #4 in today's exam prep section.)

- Let `s` be an initially empty stack.
- For each token (`tok`) in your postfix expression:
 - If `tok` is a numeric value:
 - Push `tok` onto `s`.
 - If `tok` is an operator (+, -, *, /):
 - Let `rightHandOperand` = `s.pop()`.
 - Let `leftHandOperand` = `s.pop()`.
 - Apply the given operator (`tok`) to the `leftHandOperand` and `rightHandOperand` and push the resulting value onto `s`.

- Note that the first value we pop above must be the right-hand operand, and the second must be the left-hand operand. This makes no difference with addition and multiplication, but it matters when performing subtraction and division.
- Return `s.pop()` .

For example, here's how the algorithm resolves the infix expression above $(3 + 5 * 2 - 12 / (2 * 3) - 1)$. Note that we process the tokens in that expression one-by-one, from left to right:

- 3 is numeric. Push onto stack. Current stack contents (from bottom to top): {3}
- 5 is numeric. Push onto stack. Current stack contents (from bottom to top): {3, 5}
- 2 is numeric. Push onto stack. Current stack contents (from bottom to top): {3, 5, 2}
- * is operator:
 - Pop stack for right-hand operand: 2
 - Pop stack for left-hand operand: 5
 - Perform operation: $5 * 2 = 10$
 - Push result (10) onto stack. Current stack contents (from bottom to top): {3, 10}
- + is operator:
 - Pop stack for right-hand operand: 10
 - Pop stack for left-hand operand: 3
 - Perform operation: $3 + 10 = 13$
 - Push result (13) onto stack. Current stack contents (from bottom to top): {13}
- 12 is numeric. Push onto stack. Current stack contents (from bottom to top): {13, 12}
- 2 is numeric. Push onto stack. Current stack contents (from bottom to top): {13, 12, 2}
- 3 is numeric. Push onto stack. Current stack contents (from bottom to top): {13, 12, 2, 3}
- * is operator:
 - Pop stack for right-hand operand: 3
 - Pop stack for left-hand operand: 2
 - Perform operation: $2 * 3 = 6$
 - Push result (6) onto stack. Current stack contents (from bottom to top): {13, 12, 6}
- / is operator:
 - Pop stack for right-hand operand: 6
 - Pop stack for left-hand operand: 12
 - Perform operation: $12 / 6 = 2$
 - Push result (2) onto stack. Current stack contents (from bottom to top): {13, 2}
- - is operator:
 - Pop stack for right-hand operand: 2
 - Pop stack for left-hand operand: 13
 - Perform operation: $13 - 2 = 11$
 - Push result (11) onto stack. Current stack contents (from bottom to top): {11}
- End of expression. Return sole element on stack as final result: 11

Indeed, the original infix expression that corresponds to this postfix expression, $3 + 5 * 2 - 12 / (2 * 3)$, is equal to 11.

You might be wondering where I got the postfix version of that original infix expression. It turns out there's an algorithm for converting infix expressions to postfix expressions that also uses a stack! If you're interested, you can read more about that on GeeksforGeeks: <https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/>

What's next?

On Friday, we'll examine two more ADTs: [sets](#) and [maps](#).

Following that, we'll delve into two major topics that will stick with us for the rest of the quarter: recursion and big-oh runtime analysis.

Exam Prep

1. As always, be sure to trace through all the code presented in class today to ensure that you understand how it works. Ideally, you should take a break after reviewing today's notes and then try to replicate many of those examples from scratch.

2. Be sure to review the section in today's notes labeled "(*Not mentioned in class.*)"

3. Read through the supplementary section of notes above, titled "Stack Application: Postfix Notation." Then write a function that can process postfix expressions using the stack-based algorithm presented in that section. The function should take two parameters: a string representing the postfix expression to be processed, and a reference to an integer where the final result should be stored. The function should return `true` if it successfully processes the given expression, `false` otherwise. If the function returns `false`, the reference parameter should remain unchanged. The function signature is as follows:

```
bool processPostfix(string expr, int& result);
```

For example function calls, see the test cases provided below.

For this problem, you may make the following assumptions:

- If our postfix string is valid, all tokens will be separated with a single space. For example: "25 12 * 3 + 84 _"
- If our postfix string is valid, all values will be integers, and the only operators will be `+`, `-`, `*`, and `/`

In completing this exercise, you might find the following functions from `strlib.h` useful:

- `stringSplit()`
- `stringIsInteger()`
- `stringToInteger()`

Here are some test cases to get you started:

 **Spoiler Alert!** These test cases for invalid postfix expressions contain spoilers for Exercise #4.

```

#include <iostream>
#include "console.h"
#include "SimpleTest.h"
#include "stack.h"
#include "strlib.h"
using namespace std;

PROVIDED_TEST("valid postfix expressions")
{
    int result = 0;

    EXPECT_EQUAL(processPostfix("5 10 +", result), true);
    EXPECT_EQUAL(result, 15);

    EXPECT_EQUAL(processPostfix("3 8 *", result), true);
    EXPECT_EQUAL(result, 24);

    EXPECT_EQUAL(processPostfix("5 10 12 + -", result), true);
    EXPECT_EQUAL(result, -17);

    EXPECT_EQUAL(processPostfix("10 12 + 5 -", result), true);
    EXPECT_EQUAL(result, 17);

    EXPECT_EQUAL(processPostfix("5 2 * 3 + 4 -", result), true);
    EXPECT_EQUAL(result, 9);

    EXPECT_EQUAL(processPostfix("2 3 + 4 5 + *", result), true);
    EXPECT_EQUAL(result, 45);

    EXPECT_EQUAL(processPostfix("2 3 4 * + 5 +", result), true);
    EXPECT_EQUAL(result, 19);

    EXPECT_EQUAL(processPostfix("2 3 + 4 +", result), true);
    EXPECT_EQUAL(result, 9);

    EXPECT_EQUAL(processPostfix("2 3 4 + +", result), true);
    EXPECT_EQUAL(result, 9);

    EXPECT_EQUAL(processPostfix("2 3 1 * + 9 -", result), true);
    EXPECT_EQUAL(result, -4);

    EXPECT_EQUAL(processPostfix("10", result), true);
    EXPECT_EQUAL(result, 10);
}

PROVIDED_TEST("invalid postfix expressions")
{
    int result = 0;

    EXPECT_EQUAL(processPostfix("5 10 + +", --result), false);
    EXPECT_EQUAL(result, -1);

    EXPECT_EQUAL(processPostfix("3 8 * 0 / 5 +", --result), false);
    EXPECT_EQUAL(result, -2);

    EXPECT_EQUAL(processPostfix("", --result), false);
    EXPECT_EQUAL(result, -3);

    EXPECT_EQUAL(processPostfix("10 12 + 13", --result), false);
    EXPECT_EQUAL(result, -4);

    EXPECT_EQUAL(processPostfix("10 12 13 +", --result), false);
    EXPECT_EQUAL(result, -5);

    EXPECT_EQUAL(processPostfix("10 + 20", --result), false);
    EXPECT_EQUAL(result, -6);

    EXPECT_EQUAL(processPostfix("- 10 8", --result), false);
    EXPECT_EQUAL(result, -7);

    EXPECT_EQUAL(processPostfix("- 10", --result), false);
    EXPECT_EQUAL(result, -8);

    EXPECT_EQUAL(processPostfix("-", --result), false);
    EXPECT_EQUAL(result, -9);

    EXPECT_EQUAL(processPostfix("10 15 + sandwich", --result), false);
    EXPECT_EQUAL(result, -10);
}

int main()
{
    runSimpleTests(ALL_TESTS);
    return 0;
}

```

4. What errors might we encounter while processing a postfix expression that would indicate that the expression

was invalid or malformed? Modify your solution to the previous problem to account for each of these scenarios.

Highlight for solutions:

- Encountering a token in our input string that is neither an integer nor a valid operator.
- Encountering an operator in our expression when `stack.size() < 2`. In this case, we don't have enough operands to perform the desired operation.
- Encountering division by zero.
- Encountering a final stack with `stack.size() != 1`. If there is nothing in the stack when we finish processing our expression, or if the final stack contains multiple values, the original expression must have been malformed.

5. In the test cases for invalid postfix expressions given above (in Exercise #3), `result` is being decremented with each new test case. What benefit does this have compared to simply setting `result = -1` and always checking that `result` is still -1 when returning from each of those individual test cases?

Highlight for solution: If we always check that `result` is -1 in those test cases, someone might think the goal of `processPostfix()` is to set `result` equal to -1 in the event that a postfix expression is malformed, and if they did so, they would get a false sense of security from passing all those test cases. In actuality, the function is supposed to leave that parameter completely unchanged when it encounters a malformed postfix expression. Changing `result` before we make a new call to `processPostfix()` with a malformed expression -- and checking that its value is unchanged when we return from that function -- more clearly conveys (and tests!) that desired behavior.

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-09